Spark Tips, Tricks, and Troubleshooting Hints

Notes

Intro to Apache Spark for Java and Scala Developers - Ted Malaska (Cloudera) - YouTube

- “take” doesn’t scale
- shuffle is the enemy
    - shuffle is the most expensive operation
    - exchanging data is extremely expensive
    - exchanging has to be done in lock-step
- Spark has 4 things
    - RDDs
        - Immutable
        - In-Memory (many caching options)
        - DataFrame/Dataset (RDD with a schema)
    - DAGs
        - 3 Components
            - Source
            - Transformation
                - Map
                - ReduceByKey
                - GroupByKey
                - JoinByKey
                - SpanByKey
            - Action
                - Count
                - Take
                - Foreach
        - Transformations don’t actually do anything until an action is called, they just build the DAG
    - FlumeJava APIs
    - Long-Lived Jobs
- Skew prevents systems from taking advantage of parallelism
    - How do I split my workload across all my executors?
    - If you ever see one core taking longer than your other cores, you have skew
    - You should never have skew, there is always a way to fix skew
    - Do a “hash mod” (i.e. “salt” the key to add dirt to the key space to get a more even distribution)
- Cartesian Joins suck…
    - Always a bad thing
    - When you join two tables with a many-to-many relationship, you get for each row in x, join to every row in y
        - multiplies for each join
        - doesn’t scale, distributed systems can’t handle this
    - Totally solvable
    - Use “nested structures”
        - a cell that has many rows in it
        - instead of 9 rows, you have 1 row with 3 of 1 axis and 3 of another axis
    - Use “windowing”
    - Use “reduce by key”
- You can always repartition for each stage to use the appropriate amount of parallelism for each function

---

Building Robust ETL Pipelines with Apache Spark - Xiao Li - YouTube

- In Spark there are 8 built-in data sources
    - JSON
    - CSV
    - Text
    - Hive
    - Parquet
    - ORC
    - JDBC
    - Kafka
- 3rd party data sources include:

        - Cassandra
        - MongoDB
        - Hbase
        - Avro
- Use “schema inference” for reading in semi-structured data (data lake)
- Can specify schema in code, but better to use DDL format
- Can also add .option(“header”, true) to see column names if they are included
- Can also specify spark sql to ignoreCorruptFiles = true if getting strange errors when reading in files
- Can handle corrupt individual records in 3 modes:
    - PERMISSIVE
        - puts corrupted records in a corrupted record column
        - can specify with columnNameOfCorruptRecord option
    - DROPMALFORMED
    - FAILFAST
- Can use “higher order functions” in Spark SQL to handle collections (maps, arrays, sets) in a column
    - e.g. SELECT EXISTS(values, e -> e > 30) AS v FROM table
    - Can also use:
        - TRANSFORM
        - FILTER
        - REDUCE
    - (possibly only in data bricks runtime, may be available in the future)
- Can write out parquet file with:
    - df.write.format(“parquet”) .saveAsTable("table")
- Can use spark sql to CREATE TABLE [AS SELECT]
- Unified preferred spark sql create table syntax example:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [db_name.]table_name

  [(col_name1 col_type1 [COMMENT col_comment1], ...)]

  USING datasource

  [OPTIONS (key1=val1, key2=val2, ...)]

  [PARTITIONED BY (col_name1, col_name2, ...)]

  [CLUSTERED BY (col_name3, col_name4, ...) INTO num_buckets BUCKETS]

  [LOCATION path]

  [COMMENT table_comment]

  [TBLPROPERTIES (key1=val1, key2=val2, ...)]

  [AS select_statement]
```

- New DataSource API (v2) coming with Spark 2.3
    - Better performance by fixing converting to RDD[Row] and serialize/deserialize
    - Python UDFs get faster (cool I guess)

Top 5 Mistakes When Writing Spark Applications - YouTube

- Don’t make executors too small/granular
    - Want multiple tasks to run in the same executor because it leverages caching better
- Don’t make executors that max out your worker nodes
    - Need to leave overhead for the OS/Hadoop Daemons
- 3 things to keep in mind when sizing executors (on YARN)
    - Memory Overhead
        - yarn default is 348 mb or 7% of spark.executor.memory
        - need to account for this so you don’t get oversubscribed
    - YARN Application Manager needs a core for each job running
    - HDFS throughput
        - If you have bulky executors, and you give all 16 cores to one executor, you will have bad HDFS throughput
        - Best to keep things around 4-6 (5) cores per executor to maximize HDFS throughput
- So if you start with 6 nodes, 16 cores each with 64 GB memory you’d get:
    - 17 executors, 19GB per each, 5 cores per executor
- No spark shuffle block can be larger than 2GB
    - You will get an INTEGER max value error from deep inside spark code

    - especially problematic in spark sql where default is set to 200 partitions (leads to high block size)
    - Fix by increasing number of partitions, or reducing the skew in your data
    - rdd.repartition() or rdd.coalesce() or spark.sql.shuffle.partitions = X
- How many partitions should I have?
    - 128mb per partition is a good rule of thumb
    - Don't have too few or you won't parallelize well
- Spark uses different kinds of data structures for less than 2000 partitions vs more than 2000
    - So therefore, if you're already close to 2000, bump it a bit to get better compression in memory
- Takes minutes to read a file but many hours to shuffle or join. What went wrong?
    - This is because of skew (most of the time)
    - Fix by using "salting"
- Avoid Shuffles!
- ReduceByKey over GroupByKey
- TreeReduce over Reduce
- Use Complex/Nested Types
    - use ordered + nested types

Operational Tips For Deploying Apache Spark - YouTube

- Use Parquet
- Use compression if sending lots of data over the wire
    - snappy, gzip, lzo
- The "small files" problem
    - reading up lots of small files is not efficient
    - creates lots of partitions
    - use coalesce()
- shufflePartitions is not the same as partitions
    - determines the number of tasks launched when doing a shuffle
    - increasing shuffle partitions can help with large joins
- …..?

Spark + Parquet In Depth: Spark Summit East talk by: Emily Curtin and Robbie Strickland - YouTube

- Parquet is:
    - Binary format
    - Columnar
    - Encoded
    - Compressed
    - Machine-friendly
    - API-driven
- Available to any project in the hadoop ecosystem
- Parquet tools (executable jar) to view parquet file contents as command line
- Schema structure:
    - Column Name
    - OPTIONAL | REQUIRED | REPEATED
    - Data Type
    - Encoding Info for Binary
    - Repetition Value
    - Definition Value
- (Borrows from Dremel)
- Reference Page GitHub - apache/parquet-format: Mirror of Apache Parquet
- Different encoding schemes: parquet-format/Encodings.md at master · apache/parquet-format · GitHub
- Can write and partition by whatever we want:
    - Very common to partition by time series (Year - Month - Day - etc…)
- Column chunks contain metadata about statistics (min, max, range, etc…)
- Brings in only the data you need:
    - Only the partitions you need
    - Only the columns you need
    - Only the chunks that fit your conditions

- Caveats:
    - Pushdown filtering doesn’t exactly work with object stores like S3 (no random access)
    - Pushdown filtering does not work on nested columns (yet)
- Write speed is less important because most data is write once, read forever
    - which do you want to optimize for?
- One pattern is to queue up things in Cassandra until sufficiently historical and then write them to parquet
- Other is to keep appending to existing files
    - Collect until condition is met
    - Groom collection
    - Write groomed rows to parquet (appending to existing)
- In Reality: Going back and fixing old data after it’s written is the hardest problem
    - They “IBM” have a process to go back and re-generate a set of parquet files according to some input
- Another benefit of a columnar store is that you can evolve your schema on the fly
    - in your next compressed file you’ll have another column
    - careful with schema merging as you are reading…
    - keep in mind that just because a format can handle some additions or deletions that doesn’t mean you app can handle it

____

AWS Summit Series 2016 | Santa Clara - Best Practices for Using Apache Spark on AWS - YouTube

- Tips for s3 performance with EMR and EMRFS
    - Avoid key names in lexicographical order
    - improve throughput and s3 list performance by not grouping commonly accessed data into the same bucket (and therefore key mapper). This impedes the ability to leverage parallelism
        - use hashing on prefixes or reverse the date time when you store things
    - Use columnar formats like Parquet so you’re just pushing less data over the wire
    - Compress data so you minimize bandwidth between EMR and S3
        - Make sure you use splittable compression or have each file be the optimal size for parallelization across your cluster
- Use RDS as an external Hive metastore
- One again, use columnar formats to scan less data and therefore send less data over the wire
- Use Encoders or Kyro Serialization instead of default Java serialization

____

Exceptions are the Norm: Dealing with Bad Actors in ETL: Spark Summit East talk by Sameer Agarwal - YouTube

- Why are ETL queries hard...?
    - Data can be messy (read: will be)
        - incomplete information
        - missing data stored as “”, “none”, “missing”, “n/a”, etc....
    - Data can be inconsistent (read: will be)
        - data conversion and type validation can be very error prone during construction
            - e.g. expecting a number but get “234 000”
            - european vs american date formats
        - incorrect information
            - CSV lists 5 columns, but a row does not contain 5 elements
    - Data can be constantly arriving
        - At least once vs exactly once semantics
        - Fault tolerance
        - Scalability (rate of ingress)
    - Data itself can be quite complex in practice
        - JSON can be deeply nested
        - Inconsistency in complex data magnifies the problem
- The thing that makes ETL hard is what makes it valuable
    - Downstream systems can’t easily handle bad records or corrupt data
    - There is very little ability to recover for these systems
    - Don’t deal well with heterogeneous sources
- Basic spark ETL example

spark.read.csv("/source/path")

.filter(...)

.agg(...)

.write.mode("append")

.parquet("/output/path")

- source files might be missing or corrupt
  - spark.sql.files.ignoreCorruptFiles = true
  - if set to true, job will continue to run when it finds crap and only process the good records
- records might have bad data
  - Text file formats (JSON and CSV) support 3 parse modes:
    - PERMISSIVE
      - adds a _corrupt_record column containing anything that was incorrect
      - can be configured via spark.sql.columnNameOfCorruptRecord
    - DROPMALFORMED
      - keeps on truckin'
    - FAILFAST
      - Throws a "malformed" exception in spark

Tips

- Lazy evaluations that generate an identifier combined with a union of a dataset/dataframe can lead to duplicate records being created in the final result

---

Troubleshooting

- A null pointer missing scheduler exception often means executor died

```
[Stage 0:=============>(959 + 14) / 973][Stage 5:>               (0 + 
9) / 2741]2017-10-25 13:56:55 ERROR TaskSetManager:70 - Task 0 in 
stage 5.0 failed 4 times; aborting job
Exception in thread "main" org.apache.spark.SparkException: Job 
aborted due to stage failure: Task 0 in stage 5.0 failed 4 times, most 
recent failure: Lost task 0.3 in stage 5.0 
(TID 1003, 10.132.56.26, executor 2): java.lang.NullPointerException
        at 
org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter.wri
te(UnsafeRowWriter.java:210)
        at 
org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIte
rator.processNext(Unknown Source)
        at 
org.apache.spark.sql.execution.BufferedRowIterator.hasNext(BufferedRo
wIterator.java:43)
        at 
org.apache.spark.sql.execution.WholeStageCodegenExec$$anonfun$8$$anon
$1.hasNext(WholeStageCodegenExec.scala:377)
        at 
scala.collection.Iterator$$anon$11.hasNext(Iterator.scala:408)
        at 
org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.write(Bypa
ssMergeSortShuffleWriter.java:126)
        at
```

```
org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scal
a:96)
        at
org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scal
a:53)
        at org.apache.spark.scheduler.Task.run(Task.scala:99)
        at
org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:322)
[0/1786]
        at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.
java:1149)
        at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor
.java:624)
        at java.lang.Thread.run(Thread.java:748)
Driver stacktrace:
        at
org.apache.spark.scheduler.DAGScheduler.org$apache$spark$scheduler$DA
GScheduler$$failJobAndIndependentStages(DAGScheduler.scala:1435)
        at
org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1.apply(D
AGScheduler.scala:1423)
        at
org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1.apply(D
AGScheduler.scala:1422)
        at
scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.
scala:59)
        at
scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
        at
org.apache.spark.scheduler.DAGScheduler.abortStage(DAGScheduler.scala
:1422)
        at
org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$
1.apply(DAGScheduler.scala:802)
        at
org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$
1.apply(DAGScheduler.scala:802)
        at scala.Option.foreach(Option.scala:257)
        at
org.apache.spark.scheduler.DAGScheduler.handleTaskSetFailed(DAGSchedu
ler.scala:802)
        at
org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.doOnReceive(D
AGScheduler.scala:1650)
        at
org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onReceive(DAG
Scheduler.scala:1605)
        at
org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onReceive(DAG
Scheduler.scala:1594)
```

```
        at
org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
        at
org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
        at
org.apache.spark.SparkContext.runJob(SparkContext.scala:1925)
        at
org.apache.spark.SparkContext.runJob(SparkContext.scala:1938)
        at
org.apache.spark.SparkContext.runJob(SparkContext.scala:1958)
        at
com.datastax.spark.connector.RDDFunctions.saveToCassandra(RDDFunction
s.scala:36)
        at
com.influencehealth.edh.enrich.childrens.support.PoCChangeCaptureAssi
gnment$.assignPoCChange(PoCChangeCaptureAssignment.scala:151)
        at
com.influencehealth.edh.enrich.childrens.PresenceOfChild$.assignPoC(P
resenceOfChild.scala:90)
        at
com.influencehealth.edh.enrich.childrens.PresenceOfChildBatchApp$.mai
n(PresenceOfChildBatchApp.scala:57)
        at
com.influencehealth.edh.enrich.childrens.PresenceOfChildBatchApp.main
(PresenceOfChildBatchApp.scala)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:62)
        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at
org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSub
mit$$runMain(SparkSubmit.scala:743)
        at
org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:18
7)
        at
org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:212)
        at
org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:126)
        at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
Caused by: java.lang.NullPointerException
        at
org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter.wri
te(UnsafeRowWriter.java:210)
        at
org.apache.spark.sql.catalyst.expressions.GeneratedClass$GeneratedIte
rator.processNext(Unknown Source)
        at
org.apache.spark.sql.execution.BufferedRowIterator.hasNext(BufferedRo
```

```
wIterator.java:43)
        at
org.apache.spark.sql.execution.WholeStageCodegenExec$$anonfun$8$$anon
$1.hasNext(WholeStageCodegenExec.scala:377)
        at
scala.collection.Iterator$$anon$11.hasNext(Iterator.scala:408)
        at
org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.write(Bypa
ssMergeSortShuffleWriter.java:126)
        at
org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scal
a:96)
        at
org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scal
a:53)
        at org.apache.spark.scheduler.Task.run(Task.scala:99)
        at
org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:322)
        at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.
java:1149)
        at
```

```
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor
.java:624)
        at java.lang.Thread.run(Thread.java:748)
```